From Knowledge to Noise: CTIM-Rover and the Pitfalls of Episodic Memory in Software Engineering Agents

Tobias Lindenbauer[1], **Georg Groh**[1], **Hinrich Schütze**[2]
[1]School of Computation, Information and Technology, Technical University of Munich
[2]CIS & MCML, LMU Munich
Correspondence: tobias.lindenbauer@tum.com

Abstract

We introduce CTIM-Rover[1], an AI agent for Software Engineering (SE) built on top of AutoCodeRover (Zhang et al., 2024) that extends agentic reasoning frameworks with an episodic memory, more specifically, a general and repository-level *Cross-Task-Instance Memory (CTIM)*. While existing open-source SE agents mostly rely on ReAct (Yao et al., 2023b), Reflexion (Shinn et al., 2023), or CodeAct (Wang et al., 2024), all of these reasoning and planning frameworks inefficiently discard their long-term memory after a single task instance. As repository-level understanding is pivotal for identifying all locations requiring a patch for fixing a bug, we hypothesize that SE is particularly well positioned to benefit from CTIM. For this, we build on the Experiential Learning (EL) approach ExpeL (Zhao et al., 2024), proposing a Mixture-Of-Experts (MoEs) inspired approach to create both a general-purpose and repository-level CTIM. We find that CTIM-Rover does not outperform AutoCodeRover in any configuration and thus conclude that neither ExpeL nor DoT-Bank (Lingam et al., 2024) scale to real-world SE problems. Our analysis indicates noise introduced by distracting CTIM items or exemplar trajectories as the likely source of the performance degradation.

1 Introduction

AI Agents have recently proven themselves as a competitive way of scaling test-time compute, especially in SE (Chowdhury et al., 2024). A crucial yet underexplored component of AI agents is their memory, which allows them to dynamically adapt their behavior based on prior experiences. Early approaches, such as ReAct (Yao et al., 2023b), rely on the agent's immediate trajectory or short-term memory for decision-making. Reflexion (Shinn et al., 2023) extends this by introducing long-term memory in the form of self-reflections on past failed task attempts, enabling agents to improve their reasoning and planning on a single task instance through In-Context Learning (ICL). While this yields performance gains on the current task instance, Reflexion discards these self-reflections after task completion. This results in inefficient use of computational resources and loss of valuable cross-task-instance learning opportunities. Zhao et al. (2024) address this limitation through Experiential Learning (EL), which is learning from past experiences across task instances. Their approach ExpeL achieves promising results on HotpotQA (Yang et al., 2018), WebShop (Yao et al., 2023a), and Alfworld (Shridhar et al., 2021). To better align with existing terminology, we name the memory consisting of knowledge extracted with EL "CTIM". Our work investigates whether CTIM generalizes to the more complex[2] domain of SE. We choose SE because we expect EL to be particularly valuable for uncovering the structure of a repository, reducing the number of turns taken exploring the codebase.

To adapt EL to SE we extend it to a MoEs inspired Knowledge Distillation (KD) approach that simultaneously captures high-level SE best practices and repository-specific details (e.g., project structure). We experimentally evaluate this approach by augmenting AutoCodeRover (Zhang et al., 2024) with CTIM, which we name "CTIM-Rover", and comparing the results of CTIM-Rover with those of the AutoCodeRover on a subset of SWE-bench Verified. We find that our adapted CTIM does not generalize to SE and instead degrades performance in all configurations compared to AutoCodeRover. Our detailed qualitative analysis identifies noisy CTIM items as culprits and we propose the use of embedding-based retrieval meth-

[1]https://github.com/Liqs-v2/ctim-rover

[2]CTIM-Rover's mean context is $\approx$ 4 times larger than ExpeL's on HotpotQA (Yang et al., 2018). Details in Table 3.

Proceedings of the 1st Workshop for Research on Agent Language Models (REALM 2025), pages 411–427

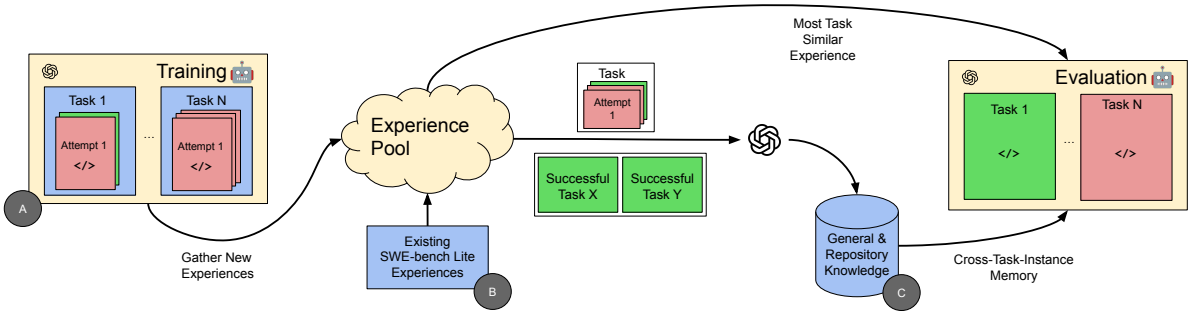

Figure 1: **CTIM-Rover Overview.** Figure inspired by ExpeL (Zhao et al., 2024). CTIM-Rover first gathers new experiences on the train set of SWE-bench Verified which we introduce in Section 3 (details in Appendix A). Then, it combines these experiences with existing experiences of AutoCodeRover (Zhang et al., 2024) on SWE-bench Lite (Jimenez et al., 2023). Next, it distills high-level and repository-level knowledge from these experiences. During evaluation, it recalls a past experience and conditions on the distilled knowledge. **Key departures from ExpeL or AutoCodeRover in blue:** (A) We extend AutoCodeRover with Reflexion (Shinn et al., 2023), allowing the agent to retry an instance up to three times while learning from its mistakes through self-reflection. (B) Compared to ExpeL, we also source experiences from past successful trajectories outside our system. (C) We introduce a novel domain-specific Knowledge Distillation (KD) phase (Figure 2) that extracts repository-level insights (e.g., common bug patterns).

ods to provide relevant, task-similar CTIMs items. The potential of this approach in the SE domain was recently demonstrated by (Su et al., 2025) who provided relevant sub-trajectories for ICL at each agent turn.

2 Related Work

2.1 Agentic Reasoning Frameworks

A core element of popular agentic reasoning frameworks (Yao et al., 2023b; Shinn et al., 2023; Wang et al., 2024) is the agent's trajectory or short-term memory, consisting of its past actions, reasoning and environment observations. Shinn et al. (2023) introduce a long-term memory consisting of self-reflections over the short-term memory of unsuccessful previous attempts. However, after concluding a task instance, existing reasoning frameworks used in SE agents do not further use the short- or long-term memory. Our work addresses this key limitation by adapting ExpeL (Zhao et al., 2024) to the SE domain.

2.2 SE Agents

SWE-agent (Yang et al., 2024) was the first openly available SE agent and leverages the ReAct reasoning framework (Yao et al., 2023b). The agent's basic search tooling combined with its interleaved bug localization and patch generation approach offers flexibility, but results in long and expensive trajectories. AutoCodeRover (Zhang et al., 2024) on the other hand, explicitly structures the task into two distinct phases: bug localization and patch generation. Additionally, it provides sophisticated search tooling during localization and constrains the patch generation phase to a maximum of three retry attempts. This ensures shorter, cost-efficient trajectories and a guaranteed termination shortly after the patch generation step. A key limitation of this approach is that the agent cannot gather additional context once it enters the patch generation phase. However, current SE agents are not yet capable of recovering from early mistakes, and their performance stagnates at later turns (Yang et al., 2025). Furthermore, neither of these agents employ CTIM. Thus, our work expands the cost-efficient AutoCodeRover with CTIM.

2.3 Concurrent Work

Lingam et al. (2024) perform self-reflection on the same task instance while prompting for a diverse set of self-reflections and additionally enhance the context with exemplar trajectories from other task instances. This approach demonstrates performance gains on programming benchmarks with comparatively short trajectories (e.g., HumanEval (Chen et al., 2021)). Especially the latter setup is closely related to CTIM-Rover with an exemplar trajectory. However, we evaluate on SWE-bench (Jimenez et al., 2023) which more closely resembles real SE tasks. Instead of abstracting from the trajectory by constructing a CTIM, (Su et al., 2025) directly retrieve synthetic sub-trajectories at

each step of the agent and achieve strong performance on SWE-bench. Furthermore, we provide the full CTIM with the user prompt at the start of an agent's trajectory instead of select subset at each turn.

3 Dataset

We use SWE-bench Verified (Chowdhury et al., 2024) without samples from the `pylint`, `astropy` and `pydata/xarray` repositories due to environment setup issues [3] as basis for our experiments. For details see Section 6. For our experiments, we rely on SWE-bench Verified, opposed to SWE-bench (Jimenez et al., 2023), as it guarantees that samples are theoretically solvable (Chowdhury et al., 2024). For the collection of past successful trajectories (Section 3.1) we use 401 samples from this benchmark and for the evaluation 45 samples.

3.1 Systematic Collection of Past Successful Trajectories

To construct a high quality CTIM, we require a diverse and representative set of successful past trajectories. These are past experiences on SWE-bench in which the agent solved an instance. This section details our systematic approach to collecting these trajectories.

To generate as many successful past trajectories as possible, we extend the baseline AutoCodeRover (Zhang et al., 2024) implementation with self-reflection capabilities. Following Shinn et al. (2023), we retry an instance up to three times and allow self-reflections to inform each subsequent attempt. While AutoCodeRover allows up to three patch generation attempts, this does not entail a complete retry on the full trajectory, nor a self-reflection between the patch generation attempts. During training we reduce the patch generation retries of AutoCodeRover from three to two to amortize some of the additional cost incurred by Reflexion retries. With this setup we gather the trajectories of 183 successfully solved instances. To further increase our training set, we supplement the collected trajectories with 53 successful AutoCodeRover trajectories from SWE-bench Lite. Because CTIM-Rover's trajectories only differ from vanilla AutoCodeRover trajectories by the addition of self-reflections, and both SWE-bench Verified and SWE-bench Lite are subsets of SWE-bench we consider this operation valid with respect to our data distribution. We use these 236 past successful trajectories to construct our CTIM. For details on their distribution see Appendix D.1.

4 Experiments

To adapt EL to SE, we extend the CTIM with a MoE (Jacobs et al., 1991) inspired repository-level CTIM (Section 4.1) and investigate ICL with successful, task-similar exemplar trajectories (Section 4.2). For distilling knowledge from trajectories, we use the reasoning model o1 (OpenAI, 2024b) because we suspect that its capabilities are beneficial when identifying pivotal agent decisions in complex SE agent trajectories (i.e., cause-effect relationships). We use GPT-4o (OpenAI, 2024a) to power the agent during training trajectory collection and the final evaluations due to budget constraints.

4.1 Cross-Task-Instance Memory (CTIM)

Our approach shares the core principle of using knowledge extracted from past successful trajectories to guide the agent on future instance with ExpeL (Zhao et al., 2024). We provide a high-level system overview of in Figure 1. To adapt this approach to SE, we extract repository-level knowledge following and conditioned on general SE knowledge in a two-phase approach detailed below (Figure 2).

Repository-Level Knowledge Distillation Our approach re-uses the KD methodology (extracting knowledge from sets of successful trajectories from distinct instances and tuples of successful and failing attempts in the same instance) and operations (add, edit, upvote or downvote[4]) introduced by Zhao et al. (2024) with the following modifications. First, we double the initial importance value of CTIM items, because we expect longer intervals between instances for which a CTIM item is applicable. This is motivated by the limited state space of ExpeL's environments, compared to the complexity of real world software repositories. Furthermore, some of our trajectories contain self-reflections. We expect these trajectories to produce especially high-quality CTIM items when extracting knowledge from tuples of successful and failing attempts in the same instance as they already contain the insights that lead to an eventual

[3] Thanks to the AutoCodeRover authors for helping us validate these.

[4] If the importance of a CTIM item falls below 0, this operation removes that item from the CTIM.

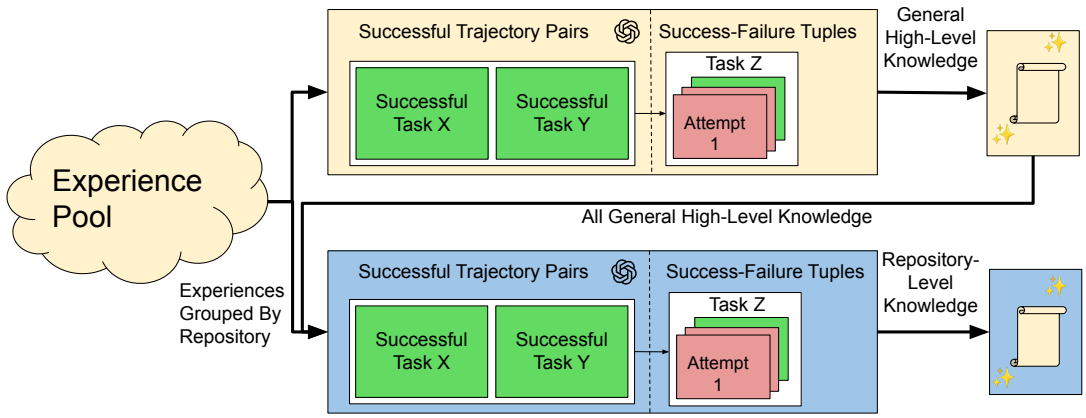

Figure 2: **CITM-Rover Knowledge Distillation (KD). Key departure from ExpeL (Zhao et al., 2024) in blue. Top:** (1) Distill generally applicable SE knowledge from pairs of successful trajectories from different task instances and (2) tuples of a successful task instance and its self-reflection retries. **Bottom:** (3) Use the generally applicable knowledge and past experience to distill repository-level knowledge from pairs of successful trajectories from different task instances within the same repository and (4) tuples of a successful task instance and its self-reflection retries for a given repository.

Configuration	Django (22)	Matplotlib (4)	Mwaskom (1)	Pytest (3)	Scikit (1)	Sphinx (5)	Sympy (9)	Overall (45)
AutoCodeRover	50	25	0	33	**100**	0	**56**	**42**
CTIM-Rover	50	**50**	0	33	**100**	0	33	40
CTIM only	36	25	0	33	**100**	0	33	31
General CTIM only	**55**	0	0	33	**100**	0	22	36
Repo-level CTIM only	41	0	0	33	**100**	0	33	31
Exemplar only	50	25	0	**67**	**100**	0	33	40

Table 1: Success rates (%) on our test set across CTIM-Rover configurations and repositories. Values in parentheses indicate the number of samples in our test set per repository.

resolution. After the first phase of general CTIM construction, we build a repository-specific CTIM by constraining all instances shown to the distilling Large Language Model (LLM) (see Section 4) to be from the same repository. Finally, we limit the maximum size of the CTIM to $c(n) = \lceil \sqrt{n} \rceil$, where n represents the number of available successful trajectories for constructing this CTIM. With this we aim to iteratively refine the CTIM to contain a concise set of high-quality insights and avoid degrading the agent's performance with noisy knowledge. For prompts see Appendix D.2, for sample CTIM items Appendix D.3.

Using the repository-level knowledge, we expect the agent will more efficiently explore its environment by re-using knowledge relating to previously explored areas of its environment. This knowledge may provide insights on (1) the structure of the project, (2) entry points or data flow and architectural patterns, (3) coding conventions encountered, (4) common failure modes relating to the application domain of the software (e.g., failure modes for image processing in OpenCV), or (5) common bugs that the agent encountered in that past.

4.2 Exemplar retrieval

In addition to providing the CTIM for ICL, we investigate if ICL with the most task-similar past successful trajectory improves performance. For this, we construct a Milvus (Wang et al., 2021a) index consisting of problem statement embeddings, using Code-T5 (Wang et al., 2021b) base as the embedding model. This model's size allows local use and it is trained for language and code, which our problem statements consist of. During evaluation, we retrieve the most task-similar past successful trajectory based on cosine similarity scores with a 90% threshold. This ensures an exemplar is only shown if a relevant one is available ($\approx$ 62% of samples).

5 Results

We evaluate CTIM-Rover's performance across the configurations listed in Table 1. CTIM-

Rover achieves only a 40% success rate, which is two percent points worse than our baseline AutoCodeRover. Surprisingly, "Exemplar only" configuration matches this performance. The "CTIM only" configuration unexpectedly degraded the performance to just 31%, 11 percent points less than the baseline. Seeing how poorly CTIM-Rover performed in the "Repo-level CTIM only" configuration, we partially attribute the performance degradation in the "CTIM only" configuration, to the repository-specific CTIM. Moreover, we observe a performance degradation even for the "django" repository, which our train set is heavily skewed towards (Figures 4 and 5). We expected instances in this repository to disproportionally benefit from the additional repository-level knowledge due to the reasons discussed in Section 4.1. Surprisingly the performance is somewhat stable compared to the baseline, even for underrepresented repositories (e.g., Pytest). This suggests source of the observed performance degradation may relate to the CTIM usage and quality rather than the quantity. We hypothesize that (1) providing all CTIM items may introduce unexpected noise because we do not filter these items for relevance regarding the instance's context, and (2) our CTIM optimization constraint leads to an overly smooth, uninformative and thus noisy CTIM. To diagnose the reasons for the poor performance, we next perform a detailed qualitative investigation of two randomly chosen samples.

5.1 Qualitative Performance Degradation Analysis

We first consider "django__django-13933", a sample that our baseline solves, but CTIM-Rover with "Repo-level CTIM only" does not. Initially, both systems invoke the correct API returning `to_python`, the function that needs a patch. However, our system decides to further investigate the `clean` function, which is also returned by the API, and does not further investigate `to_python`. This indicates an unexpected bias towards the tokens constituting "clean". In the repository-level CTIM for "django" we notice that the item in Figure 3 contains the word **clean**. Upon removing this item from the CTIM and retrying, our system correctly identifies the `to_python` function as the location for the patch and solves the sample.

Next, we focus on "django__django-15987", a sample that both AutoCodeRover and CTIM-Rover with "Repo-level CTIM only" solved, but CTIM-Rover failed to solve in the "CTIM only" configuration. The problem statement of this sample explicitly mentions the constant `FIXTURE_DIRS` and AutoCodeRover correctly searches the repository for this constant. However, CTIM-Rover with the "CTIM only" configuration does not. We notice that our CTIM does not refer to any constants and suspect that this biases our system towards lower snake-case names. Upon adding the arbitrary, capitalized item "GRANDMA LIKES PASTA" to the CTIM and retrying, our system again solves the sample. This suggests noisy CTIM biases CTIM-Rover toward suboptimal initial steps rather than helping it skip initial exploration turns and furthermore hypothesize that lengthy exemplar trajectories likely cause similar issues.

Problematic "django" CTIM Item

[...] Ensure to separate resolution from the final redirect to keep path_info **clean** while preserving the prefix in the final URL, preventing forced [...]

Figure 3: Excerpt of the repository-level CTIM item that biased our system toward investigating the incorrect `clean` function, demonstrating how seemingly innocuous knowledge can misguide the agent.

6 Conclusion

In this work we extend ExpeL (Zhao et al., 2024), which showed promising performance on HotpotQA (Yang et al., 2018), WebShop (Yao et al., 2023a), and Alfworld (Shridhar et al., 2021) to the SE domain. We introduce a repository-specific CTIM and investigate its performance on a subset of SWE-bench Verified (Chowdhury et al., 2024). Our results show that this simple EL implementation does not generalize to the SE setting and that the findings reported by Zhao et al. (2024) and Lingam et al. (2024) are thus limited to simpler environments with shorter trajectories. An extension of a general CTIM with repository-level CTIM or exemplar-based ICL does not suffice to amend this. Our investigations reveal noisy CTIM items as the likely culprit. We suggest removing the maximum size constraint from CTIM and instead focus on embedding-based retrieval of highly relevant CTIM items with respect to the problem statement. Furthermore, we suspect that the use of the CTIM should not be limited to the initial context, and instead a subset of relevant items should be provided at each turn (Su et al., 2025).

Limitations

Regarding our dataset a key limitation is that we had to remove $\approx 10\%$ of samples from SWE-bench Verified due to defective environment setup scripts. We only removed these samples after validating these issues with the AutoCodeRover authors. Furthermore, while our ablation study and qualitative analysis hint at a potential path towards a concise and focused CTIM implementation that improves, rather than degrades performance via embedding-based retrieval of CTIM items, we do not investigate such an approach. Future work may also consider the re-use of self-reflections as CTIM items more explicitly. In our work we only implicitly distill knowledge from these in the success-failure tuple setting.

Acknowledgements

We acknowledge the support of this project by Jetbrains who kindly provided API credits for our experiments. We thank the AutoCodeRover (Zhang et al., 2024) authors for the helpful discussions and support in building on top of their research. Tobias Lindenbauer thanks Sophia Schwarz for her unwavering support throughout this project.

References

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *arXiv preprint*. ArXiv:2107.03374.

Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubeh, Mia Glaese, Carlos E. Jimenez, John Yang, Leyton Ho, Tejal Patwardhan, Kevin Liu, and Aleksander Madry. 2024. Introducing swe-bench verified. Accessed on March 12, 2025.

Robert A. Jacobs, Michael I. Jordan, Steven J. Nowlan, and Geoffrey E. Hinton. 1991. Adaptive mixtures of local experts. *Neural Computation*, 3(1):79–87.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. 2023. SWE-bench: Can Language Models Resolve Real-world Github Issues?

James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, Demis Hassabis, Claudia Clopath, Dharshan Kumaran, and Raia Hadsell. 2017. Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences*, 114(13):3521–3526. Publisher: Proceedings of the National Academy of Sciences.

Vijay Lingam, Behrooz Omidvar Tehrani, Sujay Sanghavi, Gaurav Gupta, Sayan Ghosh, Linbo Liu, Jun Huan, and Anoop Deoras. 2024. Enhancing Language Model Agents using Diversity of Thoughts.

OpenAI. 2024a. Openai gpt-4o system card. Accessed on March 6, 2025.

OpenAI. 2024b. Openai o1 system card. Accessed on March 6, 2025.

Haizhou Shi, Zihao Xu, Hengyi Wang, Weiyi Qin, Wenyuan Wang, Yibin Wang, Zifeng Wang, Sayna Ebrahimi, and Hao Wang. 2024. Continual Learning of Large Language Models: A Comprehensive Survey. *arXiv preprint*. ArXiv:2404.16789 version: 2.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language Agents with Verbal Reinforcement Learning. *arXiv preprint*. ArXiv:2303.11366 [cs].

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. 2021. ALFWorld: Aligning Text and Embodied Environments for Interactive Learning. *arXiv preprint*. ArXiv:2010.03768 [cs].

Hongjin Su, Ruoxi Sun, Jinsung Yoon, Pengcheng Yin, Tao Yu, and Sercan Ö Arık. 2025. Learn-by-interact: A Data-Centric Framework for Self-Adaptive Agents in Realistic Environments. *arXiv preprint*. ArXiv:2501.10893 [cs].

Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, et al. 2021a. Milvus: A purpose-built vector data management system. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2614–2627.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024. Executable Code Actions Elicit Better LLM Agents. *arXiv preprint*. ArXiv:2402.01030 [cs].

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. 2021b. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. *arXiv preprint*. ArXiv:2109.00859.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. *arXiv preprint*. ArXiv:2405.15793.

John Yang, Kilian Leret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. 2025. SWE-smith: Scaling Data for Software Engineering Agents. *arXiv preprint*. ArXiv:2504.21798 [cs].

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W. Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. 2018. HotpotQA: A Dataset for Diverse, Explainable Multi-hop Question Answering. *arXiv preprint*. ArXiv:1809.09600.

Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. 2023a. WebShop: Towards Scalable Real-World Web Interaction with Grounded Language Agents. *arXiv preprint*. ArXiv:2207.01206 [cs].

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023b. ReAct: Synergizing Reasoning and Acting in Language Models. *arXiv preprint*. ArXiv:2210.03629 [cs].

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous Program Improvement. *arXiv preprint*. ArXiv:2404.05427.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. 2024. ExpeL: LLM Agents Are Experiential Learners. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(17):19632–19642. Number: 17.

A Our Train and Test Sets Based on SWE-bench Verified

We use the human-annotation data released with SWE-bench Verified (Chowdhury et al., 2024) for partitioning SWE-bench Verified into a train and test set. For this, we first investigate the statistical association of the `underspecified`, `false_negative` and `difficulty` features from the annotation data with the outcome of an instance being solved by any competing system on SWE-bench Verified, to inform our decision across which fields to stratify during the dataset partitioning. We base this analysis on a snapshot of the SWE-bench Verified leader board from October 30, 2024. Table 2 presents the p-values for the investigated features. We find that the `difficulty` and `false_negative` features are statistically significantly associated with the resolution of an instance at a significance level of $\alpha = 0.01$. These fields correspond to the subjective time required to resolve an instance by human annotators and whether the test suite mistakenly filters out successful solutions due to overly specific tests (e.g., requiring specific error messages), respectively. We did not find an `underspecifed` problem statement to statistically significantly affect instance resolution performance. We thus stratify across the statistically significant features to construct train and test sets that are representative of the original dataset across these success-related features.

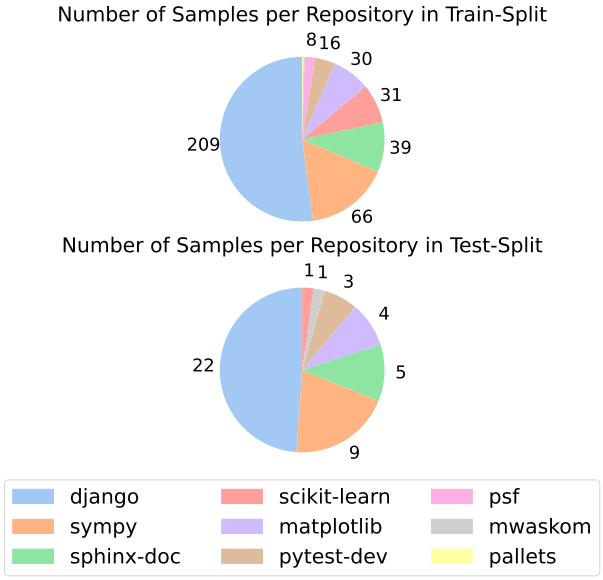

Figure 4: The distribution of repositories across our train and test sets.

In Figure 4, we show the distribution of repositories across our train and test sets and observe that data distribution of repositories decently approximated across both the train and test set even without explicitly considering this in our stratification process. However, some outliers exist (e.g., "psf" or "pallets"). For these all samples are contained in the train set. Our partitioning approach thus mostly maintains a repository overlap between train and test sets, a critical prerequisite for the cross-repository knowledge transfer of repository-level knowledge in our CTIM approach.

Feature	p-value	χ^2 statistic	Used For Stratification
`underspecified`	0.6293	0.2329	
`false_negative`	$6.3 \cdot 10^{-4}$	11.6946	✓
`difficulty`	$3.4 \cdot 10^{-4}$	18.5154	✓

Table 2: Chi-square analysis of SWE-bench Verified annotations and instance resolution status.

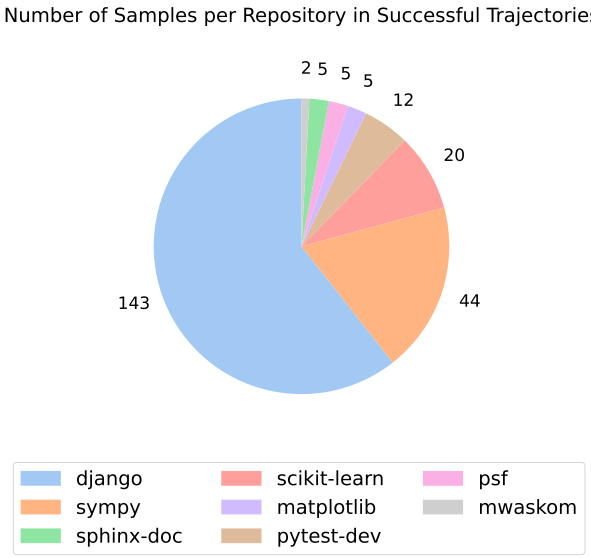

Figure 5: The distribution of repositories across successful solved instanced by CTIM-Rover on our train split and by AutoCodeRover on SWE-bench Lite (Jimenez et al., 2023). In total there are 236 solved instances based on which we create our CTIM.

B Extended Related Work

B.1 Experiential Learning (EL)

One exciting advantage of EL and CTIM is that they allow agents to continually augment their knowledge of dynamic environments through ICL. EL is loosely related to Continual Learning (CL). However, typical EL methods require updating a model's weights or architecture (Shi et al., 2024). This is often unfeasible and exposes the model to the risk of catastrophic forgetting (Kirkpatrick et al., 2017). While our implementation of EL did not improve baseline performance, the findings of Su et al. (2025) support our finding that noisy context is the cause of the performance degradation and provide an actionable way forward for tackling this issue: embedding-based retrieval of CTIM items at every agent turn.

C Extended Results

To motivate the increased complexity of SE compared to the domains investigated by ExpeL (Zhao et al., 2024) we supplement our discussion of this matter in Section 1 with statistics on turns taken and tokens consumed (Table 3. Note that the consumed tokens map to the context window at the final step in which the agent generates its patch.

D Cross-Task-Instance Memory (CTIM) Construction

In this section we provide further details on the construction of our CTIMs. In Section 5 we discuss the distribution of the trajectories that serve as basis for KD and CTIM construction in relation to our train and test set's data distributions. Then, we detail our prompt templates for CTIM creation in Section D.2. Finally, we provide sample CTIM items in Section D.3.

D.1 Distribution of Past Successful Trajectories For Knowledge Distillation (KD)

Compared to the distributions of our train and test sets in Figure 4 the available past successful trajectories for the actual CTIM construction are even more heavily skewed towards the repositories "django" and "sympy" (Figure 5). This is problematic for the general-level CTIM because it makes it challenging to distill knowledge that is generally relevant to SE. To amend this, we oversample from all repositories except "django" in the "sets of successful past trajectories" KD setting such that we show an evenly balanced number trajectories across "django" and all remaining repositories. For the repository-specific CTIM on the other hand, we expected the "django" repository to benefit from the additional knowledge in particular due to this heavy skew.

D.2 Knowledge Distillation Prompts

Figure 6 illustrates the prompt structure used in the success-failure tuple setting. Both the general and repository-level KD approaches enforce a common structural framework for the distilled knowledge and support identical operations on their respective knowledge bases. However, they dif-

Configuration	Turns				Tokens			
	Median	Mean	Min	Max	Median	Mean	Min	Max
AutoCodeRover	9	11.16	4	20	10027	11414.02	3654	31408
CTIM-Rover	7	9.62	4	20	17807	17544.93	4962	36250
CTIM only	8	10	4	20	10724	13200.96	4438	46983
General CTIM only	9	10.31	4	20	9984	12272.22	3594	37785
Repo-level CTIM only	8	9.67	4	20	10130	12139.87	3953	38730
Exemplar only	8	9.2	4	20	15579	16143.47	3365	43799

Table 3: Summary statistics for turn and token lengths on our test set ($n = 45$) across different configurations. All numeric values are rounded to two decimal places.

fer notably in terms of input configuration and focus: for general KD, the model is prompted to extract broadly applicable software engineering best practices along with common error and bug types, whereas repository-level KD explicitly targets repository-specific information—such as the repository's structure, data flow patterns, and localized characteristics. In the repository-level phase, the prompt also incorporates the previously extracted high-level general knowledge as a reference, ensuring that the new insights provide a distinct, fresh perspective. Moreover, the distilling LLM is restricted to modifying only the repository-specific knowledge base for the repository currently being processed, even though the general knowledge base remains visible solely for comparison purposes. For further details please refer to Figures 7 to 21 detailing our prompts below.

D.3 Sample CTIM Items

In this section we provide sample general and repository-level CTIM items. We provide four randomly selected general (Figure 22) and django repository-level (Figure 23) items. Additionally, we provide all repository-level CTIM items for psf (Figure 24), an underrepresented repository.

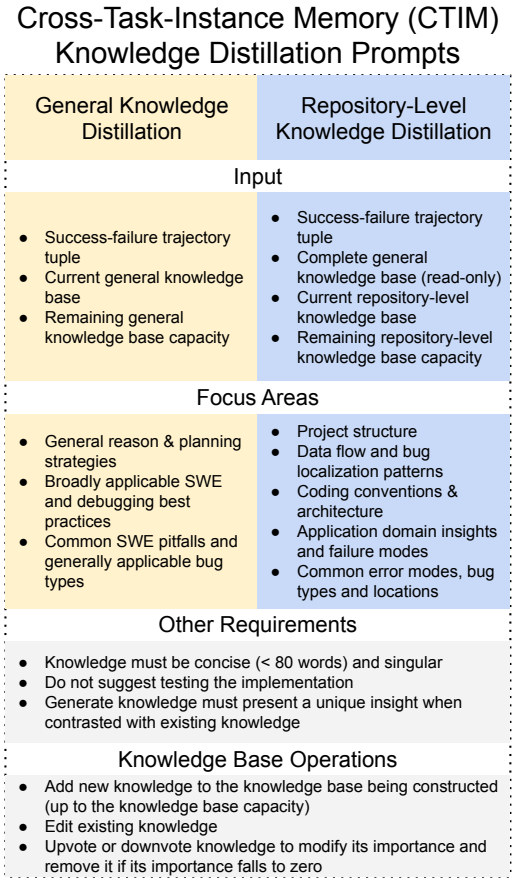

Figure 6: Key differences in our prompting strategies for general and repository-level Knowledge Distillation (KD). The general KD (left) captures broadly applicable software engineering principles, while the repository-level distillation (right) captures repository-specific patterns. In the repository-level KD, we ensure that all repository-specific data originate from the same repository. In our implementation, we refer to knowledge items as "rules" in prompting templates, but conceptually they represent distilled knowledge.

Knowledge Distillation System Prompt

```
You are an advanced reasoning agent that can ADD, EDIT, UPVOTE or DOWNVOTE rules from an existing rule
set, which is constructed by reflecting on and critiquing past successful task trajectories.
```

Figure 7: The system prompt we use for both KD phases and all KD settings.

CTIM Capacity Warning Prompt

```
You have reached the maximum ruleset size of {ruleset_cap}. The ADD  operation is now INVALID. To
reduce the ruleset size, prune low-utility rules that overlap with others by performing the
DOWNVOTE operation on them.
```

Figure 8: The CTIM capacity warning prompt we use in both KD phases and all KD settings.

CTIM Capacity Information Prompt

```
You may add up to {remaining_slots} more rules to the ruleset before reaching the maximum
of {ruleset_cap} rules.
```

Figure 9: The CTIM capacity information prompt we use in both KD phases and all KD settings.

CTIM Operations Prompt - General KD (Phase 1)

```
Provide the operations as a list
containing JSON objects of the following schema:
{{
"operation_type": {{"enum": ["ADD", "EDIT", "UPVOTE", "DOWNVOTE"]}},
"rule_id": {{"type": "integer"}},
"rule_content": {{"type": "string"}}
}}
The "operation_type" field specifies the type of operation to perform on the rule with the given
"rule_id". The "rule_id" must be an integer identifying a rule in the current
ruleset{ruleset_indices_hint}. If you are adding or editing a rule, additionally provide
the "rule_content" field with the new content of the rule.

Here is an example of a valid response:
{{"operations":
    [{{
        "operation_type": "ADD",
        "rule_content": <Extracted insight, knowledge, tip or rule>
    }},
    {{
        "operation_type": "DOWNVOTE",
        "rule_id": <Integer identifying an EXISTING rule that is contradictory to
                    another rule, this sample or too similar to another rule>
    }},
    {{
        "operation_type": "EDIT",
        "rule_id": <Integer identifying an EXISTING rule>,
        "rule_content": <Extracted insight, knowledge, tip or rule to update and
                         enhance the EXISTING rule with>
    }}]
}}
```

Figure 10: The CTIM operations prompt we use in the high level KD phase and both its KD settings.

CTIM Operations Prompt - General KD (Phase 1) continued

```
Do not mention the trajectories or their ids explicitly in your responses. Do not reference specific
file, class, function or variable names to ensure that your ruleset is general and transferable to
other task instances and repositories. You can use any of the valid operation types multiple times.

Each existing rule can be modified only once.
The following operations are valid:
- UPVOTE an EXISTING rule if it is strongly relevant in your current context and
  trajectories. Valid fields: [operation_type, rule_id]
- DOWNVOTE an EXISTING rule if the rule contradicts your current context and
  trajectories or is similar to or a duplicate
of another existing rule. Make use of this operation to achieve a concise ruleset
that is relevant across repositories and task instances.
If you downvote a rule often enough it will be removed from the ruleset. Valid
fields: [operation_type, rule_id]
- EDIT an EXISTING rule if it is not general enough or could be enhanced given your
  current context by rewriting, adding or removing content. Valid fields:
  [operation_type, rule_id, rule_content]
- ADD a NEW rule if you identified insights that are generally applicable and
  transferable to other task instances. Make sure that the new rule is distinct
from existing rules. Valid fields: [operation_type, rule_content]

Key requirements:
- The only operation that is valid on rules that do not yet exist is ADD.
- If you have reached the maximum ruleset size, you must not add any new rules.
  Instead, you must edit existing rules or upvote/downvote existing rules.
- You may provide between 1 and 4 operations.
```

Figure 11: The CTIM operations prompt we use in the high level KD phase and both its KD settings continued.

Success-Failure Trajectory KD Setting - General KD (Phase 1)

```
You are given a set of successful task trajectories that relate to fixing bugs in open-source code
repositories. During these trajectories you correctly identified the location of the buggy code,
wrote a patch which fixed the bug in the code and passed all test cases, meaning you also didn't in
introduce any new bugs.

Below follow the past successful task trajectories. The set of trajectories is delimited by the
<PAST_SUCCESSFUL_TRAJECTORIES> and </PAST_SUCCESSFUL_TRAJECTORIES> tags. Each trajectory is
wrapped by the <TRAJECTORY-i> and </TRAJECTORY-i> tags, where i identifies the i-th trajectory
in the set below:
<PAST_SUCCESSFUL_TRAJECTORIES>
{past_successful_trajectories}
</PAST_SUCCESSFUL_TRAJECTORIES>

Next, follow a set of rules that you have extracted so far. The ruleset is limited to {ruleset_cap}
rules. Any rules beyond {ruleset_cap} rules will be ignored:
{current_ruleset}
{remaining_slots_information}

By examining the successful trajectories, and the existing rules above you should update the
existing ruleset by adding, editing, upvoting or downvoting rules. The resulting ruleset must
consist of high-level knowledge, insights or tips that are generally applicable, covering
the following aspects:
1. Reasoning and planning strategies that serve as guiding signals for future task attempts,
especially with respect to identifying the locations of buggy code effectively.
2. Coding practices, patterns, and idioms that are generally applicable to writing high-quality,
staff senior level code, to fix bugs.
3. Common pitfalls and error patterns in software engineering that are relevant to identifying
and fixing buggy code.
```

Figure 12: The prompt describing the success-failure trajectory pair KD setting for the high level KD phase.

Success-Failure Trajectory KD Setting - General KD (Phase 1) continued

```
Key requirements for rules:
- DO NOT suggest testing the implementation. The agent using your ruleset is UNABLE to test
its implementation. It must generate a correct patch on the first attempt.
- Generated rules must be concise (less than 80 words) and should be focused on a single,
specific aspect or insight.
- Generated rules must be unique with respect to other, existing rules and contribute a new,
unique piece of information, knowledge or perspective.

This ruleset should serve as the basis for guiding future task attempts in locating and
fixing bugs to a successful completion and empower the agent to improve its planning,
reasoning, coding skills, bug localization skills.
```

Figure 13: The prompt describing the success-failure trajectory pair KD setting for the high level KD phase continued.

Sets of Successful Trajectories KD Setting - General KD (Phase 1)

```
Below you will find multiple past attempts at fixing a bug in an open-source code repository.
The first few trajectories show failed attempts, the last trajectory shows a successful bug fix.
All attempts are related to fixing the same bug in the same codebase. Compare and contrast the
successful and failed attempts to understand why the initial attempts failed and which change
in the reasoning, planning, coding or bug localization strategy could have led to a correct
patch generation in the first attempt. Consider the self-reflections that took place between
the failed attempts to understand which changes were made in the reasoning, planning, coding
or bug localization strategy that led to the bug being fixed in the last trajectory.

Below follow the task attempts denoted by <FAILED_TASK_ATTEMPT-i> and </FAILED_TASK_ATTEMPT-i> tags
where i identifies the i-th failed attempt and the successful task attempt is denoted by the
<SUCCESSFUL_TASK_ATTEMPT> and </SUCCESSFUL_TASK_ATTEMPT> tags. Only failed task attempts contain
a self-reflection:
{success_failure_trajectory}

Next, follow a set of rules that you have extracted so far. The ruleset is limited to {ruleset_cap}
rules. Any rules beyond {ruleset_cap} rules will be ignored:
{current_ruleset}

{remaining_slots_information}

By examining and comparing the successful and failed attempts, and the existing rules above you
should update the existing ruleset by adding, editing, upvoting or downvoting rules. The resulting
ruleset must consist of high-level knowledge, insights or tips that are generally applicable,
covering the following aspects:
1. Reasoning and planning strategies that serve as guiding signals for future task attempts,
especially with respect to entifying the locations of buggy code effectively.
2. Coding practices, patterns, and idioms that are generally applicable to writing high-quality,
staff senior level code, to fix bugs.
3. Common pitfalls and error patterns in software engineering that are relevant to identifying
and fixing buggy code.

Key requirements for rules:
- DO NOT suggest testing the implementation. The agent using your ruleset is UNABLE to test
its implementation. It must generate a correct patch on the first attempt.
- DO NOT suggest reflecting on a past trajectory or attempt. The agent using your ruleset
is UNABLE to reflect on a past trajectory or attempt. It must generate a correct patch on
the first attempt.
```

Figure 14: The prompt describing the sets of successful trajectories KD setting for the high level KD phase.

Sets of Successful Trajectories KD Setting - General KD (Phase 1) continued

```
- Generated rules must be concise (less than 80 words) and should be focused on a single,
specific aspect or insight.
- Generated rules must be unique with respect to other, existing rules and contribute a
new, unique piece of information, knowledge or perspective.

This ruleset should serve as the basis for guiding future task attempts in locating and fixing
bugs to a successful completion.
It should empower the agent to improve its planning, reasoning, coding, and bug localization
skills.
```

Figure 15: The prompt describing the sets of successful trajectories KD setting for the high level KD phase continued.

CTIM Operations Prompt - Repository-Level KD (Phase 2)

```
Provide the operations as a list containing JSON objects of the following schema:
{{
"operation_type": {{"enum": ["ADD", "EDIT", "UPVOTE", "DOWNVOTE"]}},
"rule_id": {{"type": "integer"}},
"rule_content": {{"type": "string"}},
"knowledge_type": {{"enum": ["repository_structure", "architectural_pattern",
"coding_convention", "error_pattern",
"application_domain"]}}
}}
The "operation_type" field specifies the type of operation to perform on the rule with the given
"rule_id". The "rule_id" must be an integer identifying a rule in the current
ruleset{ruleset_indices_hint}. If you are adding or editing a rule, additionally
provide the "rule_content" field with the new content of the rule. If you are adding
a rule, you must also specify the "knowledge_type" of the rule.

Here is an example of a valid response:
{{"operations":
    [{{
        "operation_type": "ADD",
        "rule_content": <Extracted insight, knowledge, tip or rule>,
        "knowledge_type": "error_pattern"
    }},
    {{
        "operation_type": "ADD",
        "rule_content": <Knowledge about the application domain of the project and typical
                         edge cases resulting from this.>
        "knowledge_type": "application_domain"
    }},
    {{
        "operation_type": "DOWNVOTE",
        "rule_id": <Integer identifying an EXISTING rule that is contradictory to another
                    rule, this sample or too similar to another rule>
    }},
    {{
        "operation_type": "EDIT",
        "rule_id": <Integer identifying an EXISTING rule>,
        "rule_content": <Extracted insight, knowledge, tip or rule to update and enhance
                         the EXISTING rule with>
    }}]
}}
```

Figure 16: The CTIM operations prompt we use in the repository-level KD phase and both its KD settings.

CTIM Operations Prompt - Repository-Level KD (Phase 2) continued

```
Do not mention the trajectories or their ids explicitly in your responses. You may reference specific
file, class, function names, but keep in mind that the repository evolves over time and files,
classes or functions may be renamed, removed or refactored. You can use any of the valid operation
types multiple times. Each existing rule can be modified only once. The following operations
are valid:

- UPVOTE an EXISTING rule if it is strongly relevant in your current context and trajectories.
  Valid fields: [operation_type, rule_id]
- DOWNVOTE an EXISTING rule if the rule contradicts your current context and trajectories or it
  is similar to or a duplicate of another existing rule (including general purpose rules).
  Make use of this operation to achieve a concise ruleset that is relevant across
  repositories and task instances. If you downvote a rule often enough it will be
  removed from the ruleset. Valid fields: [operation_type, rule_id]
- EDIT an EXISTING rule if it is not general enough or could be enhanced given your current
  context by rewriting, adding or removing content. Valid fields: [operation_type, rule_id,
  rule_content]
- ADD a NEW rule if you identified insights that are generally applicable and potentially
  beneficial to other task instances in the same repository. Make sure that the new rule is
  unique. Valid fields: [operation_type, rule_content, knowledge_type]

Key requirements:
- The only operation that is valid on rules that do not yet exist is ADD.
- If you have reached the maximum ruleset size, you must not add any new rules. Instead, you
  must edit existing rules or upvote/downvote existing rules.
- You may provide between 1 and 4 operations.
```

Figure 17: The CTIM operations prompt we use in the repository-level KD phase and both its KD settings. Continued.

Success-Failure Trajectory KD Setting - Repository-Level KD (Phase 2)

```
You are given a set of successful task trajectories that relate to fixing issues the real-world
repository '{repository_name}'. During these trajectories you correctly identified the location
of the buggy code, wrote a patch which fixed the bug in the code and passed all test cases,
meaning you also didn't in introduce any new bugs. Due to the natural evolution of software over
time the state of the repository when you carried out the tasks in the example trajectories below
may differ slightly. You might encounter differences with respect to the project structure, and
file, class, method or variable names. If you encounter conflicting information, do not record
any rules regarding the conflicting elements.

Below follow the past successful task trajectories. The set of trajectories is delimited by the
<PAST_SUCCESSFUL_TRAJECTORIES> and </PAST_SUCCESSFUL_TRAJECTORIES> tags. Each trajectory is
wrapped by the <TRAJECTORY-i> and </TRAJECTORY-i> tags, where i identifies the i-th trajectory
in the set below:
<PAST_SUCCESSFUL_TRAJECTORIES>
{past_successful_trajectories}
</PAST_SUCCESSFUL_TRAJECTORIES>

Next, follows the frozen set of high-level, general purpose rules that you have extracted previously.
These rules are READ-only, you must not perform any operations on them. You may refer to these rules
directly in the repository level rules as 'GENERAL PURPOSE RULE-i' to highlight their specific
application, knowledge gaps or discrepancies with respect to the current repository:
{general_ruleset}
```

Figure 18: The prompt describing the sets of successful trajectories KD setting for the repository-level KD phase.

Success-Failure Trajectory KD Setting - Repository-Level KD (Phase 2) continued

```
Below follows the modifiable set of repository-level rules that you have extracted so far. The
repository-level ruleset is limited to {ruleset_cap} rules. Any rules beyond {ruleset_cap} rules
will be ignored:
{current_repository_level_ruleset}

By examining the successful trajectories, and the existing general purpose and repository-level
rules above you should update the repository-level ruleset by adding, editing, upvoting or downvoting
repository-level rules. The resulting ruleset must consist of repository-specific knowledge,
insights or tips that are unique to this codebase and provide new insights that are distinct from the
general purpose rules. Repository-level rules may cover the following aspects:
1. Repository-level bug localization and environment exploration patterns that help locate relevant
code sections quickly, including key file locations, module relationships.
2. Repository-level coding conventions, architectural principles, design patterns, and
implementation approaches that are consistently used across the codebase and should be followed
when making changes.
3. Repository-level error or exception handling strategies, including custom errors or exceptions
4. The application domain of the project (e.g., Does the software handle images or text and what
kind? Is it a command line application or does it have a GUI? Does it handle HTTP requests? Is it a
highly technical, mathematical application?)
5. Common edge cases or failure modes related to the project's specific application domain. What
are common errors or potential pitfalls in these application domains?).

Key requirements for rules:
- DO NOT suggest testing the implementation. The agent must generate correct patches on the first
attempt by leveraging general and repository-specific rules identified above.
- Generated rules must be concise (less than 80 words) and should be focused on a single, specific
aspect or insight.
- Generated rules must be unique with respect to other, existing rules and contribute a new, unique
piece of information, knowledge or perspective.

This ruleset serves as the basis for guiding future task attempts within this repository in locating
and fixing bugs to a successful completion. It should empower the agent to improve its planning,
reasoning, coding, and bug localization skills.

{remaining_slots_information}
```

Figure 19: The prompt describing the sets of successful trajectories KD setting for the repository-level KD phase. Continued.

Success-Failure Trajectory KD Setting - Repository-Level KD (Phase 2)

```
Below you will find multiple past attempts at fixing a bug in an open-source code repository. The
first few trajectories show failed attempts, the last trajectory shows a successful bug fix.
All attempts are related to fixing the same bug in the same codebase. Compare and contrast the
successful and failed attempts to understand why the initial attempts failed and which change in
the reasoning, planning, coding or bug localization strategy could have led to a correct patch
generation in the first attempt.Consider the self-reflections that took place between the failed
attempts to understand which changes were made in the reasoning, planning, coding or bug localization
strategy that led to the bug being fixed in the last trajectory.

Below follow the task attempts denoted by <FAILED_TASK_ATTEMPT-i> and </FAILED_TASK_ATTEMPT-i> tags
where i identifies the i-th failed attempt and the successful task attempt is denoted by the
<SUCCESSFUL_TASK_ATTEMPT> and </SUCCESSFUL_TASK_ATTEMPT> tags. Only failed task attempts contain a
self-reflection:
{success_failure_trajectory}

Next, follows the frozen set of high-level, general purpose rules that you have extracted previously.
These rules are READ-only, you must not perform any operations on them. You may refer to these rules
```

Figure 20: The prompt describing the success-failure trajectory pair KD setting for the repository-level KD phase.

Success-Failure Trajectory KD Setting - Repository-Level KD (Phase 2) continued

```
directly in the repository level rules as 'GENERAL PURPOSE RULE-i' to highlight their specific
application, knowledge gaps or discrepancies with respect to the current repository:
{general_ruleset}

Below follows the modifiable set of repository-level rules that you have extracted so far. The
repository-level ruleset is limited to {ruleset_cap} rules. Any rules beyond {ruleset_cap} rules
will be ignored:
{current_repository_level_ruleset}
```

Figure 21: The prompt describing the success-failure trajectory pair KD setting for the repository-level KD phase. Continued.

Sample General CTIM Items

```
- Perform targeted input validations, ensuring each parameter or feature aligns with immediate needs
and preventing unwanted callability, type-mismatch, or boundary issues.

- Examine error messages to locate the failing logic. Also confirm if the framework\u2019s checks
might be incomplete or incorrectly flag valid usage, especially for advanced lookups or
edge cases.

- Always confirm that referenced methods or variables exist, are spelled correctly, remain valid,
and that decorators or partials do not obscure them.

- Focus changes on the minimal relevant locations, referencing existing methods or design
patterns to maintain consistency, reduce duplication, and lower risk of new bugs.
```

Figure 22: Four random general CTIM samples.

Sample Django Repository-Level CTIM Items

```
- When refactoring special-case or zero-quantity paths in the app\u2019s code
(like max_post_process_passes=0), skip irrelevant steps entirely to avoid referencing uninitialized
variables. If the field or setting indicates no passes or empty states, ensure the logic short-circuits
properly. This avoids spurious errors from referencing variables that never get assigned.

- When unregistering or registering custom lookups in RegisterLookupMixin, always call
_clear_cached_lookups afterward to avoid stale lookup references and maintain consistency
with register_lookup.

- When retrieving fields from database insert operations using returning_fields, ensure
from_db_value or equivalent logic is consistently applied, matching standard retrieval.
This prevents raw values from bypassing normal conversions, especially for custom fields
that rely on from_db_value to transform them into appropriate Python objects.

- When customizing admin logic in Django, including catch-all or fallback views, confirm usage of
request.path vs request.path_info. request.path preserves the script name prefix required in certain
redirects, while request.path_info omits it. Ensure to separate resolution from the final redirect
to keep path_info clean while preserving the prefix in the final URL, preventing forced script
name issues. Additionally, ensure admin checks referencing placeholders or fields include
the actual field name in error messages for clarity.
```

Figure 23: Four random django CTIM samples.

All psf Repository-Level CTIM Items

- In 'prepare_headers' (requests/models.py::PreparedRequest), headers with a None value become the literal 'None' string if not filtered out. Always remove such keys to avoid invalid headers.

- When unregistering or registering custom lookups in RegisterLookupMixin, always call _clear_cached_lookups afterward to avoid stale lookup references and maintain consistency with register_lookup.

- When retrieving fields from database insert operations using returning_fields, ensure from_db_value or equivalent logic is consistently applied, matching standard retrieval. This prevents raw values from bypassing normal conversions, especially for custom fields that rely on from_db_value to transform them into appropriate Python objects.

- When customizing admin logic in Django, including catch-all or fallback views, confirm usage of request.path vs request.path_info. request.path preserves the script name prefix required in certain redirects, while request.path_info omits it. Ensure to separate resolution from the final redirect to keep path_info clean while preserving the prefix in the final URL, preventing forced script name issues. Additionally, ensure admin checks referencing placeholders or fields include the actual field name in error messages for clarity.

Figure 24: All psf repository-level CTIM samples.